Lightweight Blockchain Logging for Data-Intensive Applications

Yuzhe (Richard) Tang[1(✉)], Zihao Xing[1], Cheng Xu[2], Ju Chen[1], and Jianliang Xu[2]

[1] Syracuse University, Syracuse, NY, USA
{ytang100,zixing,jchen133}@syr.edu
[2] Hong Kong Baptist University, Kowloon Tong, Hong Kong
{chengxu,xujl}@comp.hkbu.edu.hk

Abstract. With the recent success of cryptocurrency, Blockchain's design opens the door of building trustworthy distributed systems. A common paradigm is to repurpose the Blockchain as an append-only log that logs the application events in time order for subsequent auditing and query verification. While this paradigm reaps the security benefit, it faces technical challenges especially when being used for data-intensive applications.

Instead of treating Blockchain as a time-ordered log, we propose to lay the log-structured merge tree (LSM tree) over the Blockchain for efficient and lightweight logging. Comparing other data structures, the LSM tree is advantageous in supporting efficient writes while enabling random-access reads. In our system design, only a small digest of an LSM tree is persisted in the Blockchain and minimal store operations are carried out by smart contracts. With the implementation in Ethereum/Solidity, we evaluate the proposed logging scheme and demonstrate its performance efficiency and effectiveness in cost saving.

1 Introduction

Recent years witnessed the advent and wide adoption of the first cryptocurrency, BitCoin [3], followed by many others including Ethereum [4], Litecoin [8], Namecoin [19], etc. The initial success of cryptocurrency demonstrates the trustworthiness of Blockchain, the underlying platform of cryptocurrency. The Blockchain supports the storage and processing of cryptocurrency transactions. In abstraction, it is a trust-decentralized network storing transparent state designed with incentives to enable open membership at scale. A line of the latest research and engineering aims at applying the trustworthy design of Blockchain for applications beyond cryptocurrency.

A common paradigm of repurposing Blockchain is to treat the Blockchain as a public append-only log [23], where application-level events are logged into the Blockchain in the order of time, and the log is used later for verification and auditing. While this public-log paradigm reaps the security benefit of Blockchain, it is limited to the applications handling small data

A. Zohar et al. (Eds.): FC 2018 Workshops, LNCS 10958, pp. 308–324, 2019.
https://doi.org/10.1007/978-3-662-58820-8_21

(due to high Blockchain storage cost) and tolerating long verification delay (linear scanning the entire chain for verification).

In this work, we tackle the research of repurposing Blockchains for hardening the security of data-intensive applications hosted in a third-party platform (e.g., cloud). A motivating scenario is to secure the cloud-based Internet-of-things (IoT) data storage where the IoT data producers continuously generate an intensive stream of data writes to the third-party cloud storage which serves data consumers through queries. Including Blockchain could enhance the trustworthiness of the third-party cloud storage.

A baseline approach is to log the sequence of data writes in the time order into the Blockchain, in a similar way to log-structured file systems [27]. This approach causes a high read latency (linear to the data size). Another baseline is to digest the latest data snapshot, e.g., using Merkle tree, and place the digest inside the Blockchain. In the presence of dynamic data, the digest scheme usually follows classic B-tree alike data structures [17,28] that perform "in-place" updates. These schemes incur high write amplification as writing a record involves a read-modify-write sequence on the tree and has $O(logN)$ complexity per write. On Blockchain, this high write amplification causes high cost, as writing a data unit in Blockchain is costly (which involves duplicated writes on miners and expensive proof-of-work alike computation). The problem compounds especially in the write-intensive applications as IoT streams.

To log write-intensive applications using write-expensive Blockchain, we propose to place the log-structured merge tree (LSM tree) [24] over the Blockchain for efficient and lightweight logging. An LSM tree is a write-optimized data structure which supports random-access reads; comparing the above two baselines (append-only log as in log-structured file system and update-in-place structures in database indices), an LSM tree strikes a better balance between read and write performance and is adopted in many modern storage systems, including Google BigTable [15]/LevelDB [7], Apache HBase [2], Apache Cassandra [1], Facebook RocksDB [5], etc.

At a high level, an LSM tree lays out its storage into several "levels" and supports, in addition to reads/writes, a compaction operation that reorganizes the leveled storage for future read/write efficiency. We propose a scheme to log the LSM tree in Blockchain: (1) individual levels are digested using Merkle trees with the 128-bit root hashes stored in Blockchain. (2) The compaction that needs to be carried out in a trustworthy way is executed in smart-contracts, which allow for computations on modern Blockchain, such as Ethereum [4]. Concretely, we propose compaction mechanisms that realize several primitives inside the smart contract. We propose a duplicated compaction paradigm amenable for implementation on the asynchronous Smart-Contract execution model in Ethereum. Based on the primitives and paradigm, we realize both sized and leveled compaction mechanisms in the smart contract.

We have implemented the design on Ethereum leveraging its Smart-contract language, Solidity [10], and programming support in the Truffle framework [11]. In particular, invoking a Solidity smart-contract is asynchronous and our system

addresses this property by asynchronously compacting the LSM storage. Based on the implementation, we evaluate the cost of our proposed scheme with the comparison to alternative designs. The results show the effectiveness of cost-reducing approaches used in our work.

The contributions of this work are the following:

1. We propose TPAD, a novel architecture to secure outsourced data storage over the Blockchain. The TPAD architecture considers an LSM-tree-based storage protocol and maps security-essential state and operations to the Blockchain. The architecture includes a minimal state in Blockchain storage and offline compaction operations in the asynchronous smart-contract.
2. We implement a prototype on Ethereum/Solidity that realizes the proposed design. Through evaluation, we demonstrate the effective cost saving of the TPAD design with the comparison to state-of-the-art approaches.

The rest of the paper is organized as following: Section 2 formulates the research problem. The proposed technique, LSM-tree based storage over the Blockchain, is presented in Sect. 3. The system implementation is described in Sect. 4. Evaluation is presented next in Sects. 5 and 7 surveys the related work. Section 8 concludes the paper.

2 Problem Formulation

2.1 Target Applications

This work targets the application of secure data outsourcing in a third-party host (e.g., Amazon S3). A particular scenario of interest is to outsource the data generated by the Internet of things (IoT) devices to the cloud storage, which serves read requests from data consuming applications. The IoT data is usually personal and could be security sensitive; for instance, in the smart home, the IoT devices such as smart TV controller capture residents' daily activities which could reveal personal secrets such as TV view habits. The IoT data, on the other hand, can be used to improve the life quality and enable novel applications. For instance, analyzing patient's activities at home can improve out-patient care and predict possible disease. In practice, various IoT data is widely collected and outsourced [16]. A noteworthy characteristic of our target application is that data is generated continuously and intensively. The workload is more write intensive than the static workload (e.g., in classic database systems).

2.2 System Model

The data-outsourcing system consists of data producers, a cloud host, and several data consumers. A data producer submits data write requests to the cloud and a data consumer submits data read requests to the cloud. The cloud exposes a

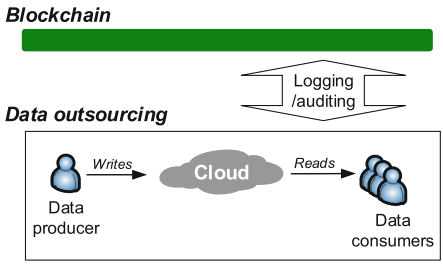


Fig. 1. Logging data outsourcing in Blockchain

standard key-value store interface for reads and writes. Formally, given key k, value v, timestamp ts, a data write and data read are described below:

$$\begin{aligned} ts &:= \mathsf{Put}(k, v) \\ \langle k, v, ts \rangle &:= \mathsf{Get}(k, ts_q) \end{aligned} \tag{1}$$

In our system, we assume the data producers and consumers are trusted. The third-party cloud is untrusted and it can launch various attacks to forge an answer to the consumer which will be elaborated on in Sect. 2.3.

Our data-outsourcing system has a companion of Blockchain, as illustrated in Fig. 1. The Blockchain logs certain events in the workflow of data outsource for the purpose of securing it.

2.3 Security Goals

In the presence of the untrusted host, there are threats that could compromise data security. An adversary, be it the cloud host or man-in-the-middle adversary in networks, could forge a fake answer to a data consumer and violate the data integrity, membership authenticity, etc. The data integrity can be protected by simply attaching a message-authentication code (MAC) to each key-value record. This work considers the more advanced attacks — membership attacks that manifest in many forms: It could be the untrusted host deliberately skips query results and presents an incomplete answer (violating query completeness). It could be the host presents a stale version of the answer (violating query freshness). It could have the host to return different answers to different consumers regarding the same query (violating the fork consistency). On the write path, a man-in-the-middle adversary could replay a write request to result in incorrectly duplicated data versions. The formal definition of membership data authenticity is described in the existing protocols of authenticated data structures (ADS) [21,26,29,32].

While this work mainly focuses authenticity, we consider a weak security goal w.r.t. the data confidentiality that deterministic encryption suffices. The extension for data confidentiality will be discussed in Sect. 6.

2.4 Existing Techniques and Applicability

Existing works on **ADS construction**, while ensuring the security of membership authentication, are mostly designed based on read-optimized database structures such as B trees and R trees [17,28] that perform data updates in place. These update-in-place structures translate an update operation from applications to a read-modify-write sequence on the underlying storage medium, and they are unfriendly to the write performance. The only ADS work we are aware of on address write efficiency is [25], which is however constructed using expensive lattice-based cryptography.

Without security, there are various write-optimized **log-structured data structures** that do not perform in-place updates but conduct append-only writes instead. A primary form of these data structures is to organize the primary data storage into a time-ordered log of records where an update is an append to the log end and a read may have to scan the entire log. The pure-log design is widely used in the log-structured journaling file systems [27].

A log-structured merge tree [24] represents a middle ground between the read-optimized update-in-place structure and the write-optimized log. An LSM tree serves a write in an append-only fashion and also supports random-access read without scanning the entire dataset. The LSM design has been adopted in many real-world cloud storage systems, including Google BigTable [15]/LevelDB [7], Apache HBase [2], Apache Cassandra [1], Facebook RocksDB [5], etc. The read-write characteristic of an LSM tree renders it well suited for the applications of IoT data outsourcing.

2.5 Motivation

Our target applications such as IoT data outsourcing feature a high-throughput stream of data updates and random-access read queries. As aforementioned, a Log-Structured Merge Tree is a good fit for this workload, assuming some offline hours for data compaction.

To map the LSM-tree workflow in an outsourcing scenario, it is essential to find a trusted third-party to conduct the data-compaction work. Relying on one of data owners to do the compaction is unfeasible due to availability, data owner's limited power (e.g., a low-end IoT device), etc.

We propose to leverage the Blockchain for the secure compaction in LSM storage. The decentralized design and large-scale deployment of existing Blockchain render it a trustworthy platform. The new smart-contract interface of the latest Blockchain makes it friendly to run general-purpose trustworthy computation on the platform.

Despite the advantages, designing a system for Blockchain-based LSM-storage outsourcing is non-trivial. Notably, Blockchain's innate limitation (in low storage capacity, high cost, low write throughput) presents technical challenges when being adapted to the high-throughput data-outsourcing workflow. We address these challenges by limiting Blockchain's involvement in the online path of data outsourcing, such that the state on Blockchain can be "updated" infrequently.

2.6 Preliminary: LSM Trees

The mechanism of an LSM tree is the following: It represents a dataset by multiple sorted runs (or files) and organized in several so-called "levels". The first level stores the most recent data writes and is "mutable". Other levels are immutable and are updated only in an offline manner. Concretely, a data write synchronously updates the first level. The first level may periodically persist data to an external place, called write-ahead log (WAL). When the first level becomes full, it is flushed to the next level. A read iterates over levels, and for each level, it is served by an indexed lookup. In the worst case, a read has to scan all levels and in practice, the total number of levels is bounded. In addition, if the application exhibits some data locality (i.e., reads tend to access recently updated data), a read can stop in the first couple of levels. An LSM tree supports a compaction[1] operation that merges multiple sorted runs into one and helps reorganizes the storage layout from a write-optimized one to a read-optimized one. The compaction is a batched job that usually runs asynchronously and during offline hours. There are two flavors in compaction, namely, flush and merge. A flush operation takes as input multiple sorted runs at level i and produces a sorted run as output at level $i+1$. A merge operation takes as input one selected file at level i and multiple files at level $i+1$ that overlap the selected file in key ranges. It produces sorted runs that replace these input files at level $i+1$.

An LSM mechanism supports different policies to trigger the execution of a compaction. These policies include sized configuration and leveled configuration: (1) In a sized configuration, each tree level has the capacity of storing a fixed number of sorted files, say K. The file size at level i is K^i (the first level has i to be 0). A flush-based compaction is triggered when there are K files filled in a level, say i. The compaction merges all K files at level i into one file at level $i+1$. With the sized-compaction policy, files at the same level may have their content overlap in key ranges, and a read has to scan all files in a level. (2) In a leveled configuration, any tree level is a sorted run where different files do not overlay in their key ranges. Data at level 0 is flushed to level 1 and data at level i, $\forall i \geq 1$, is merged to level $i+1$ [7]. A compaction can be triggered by application-specific conditions. A read within a level can be served by an indexed lookup without scanning.

3 LSM Data Storage over Blockchain

3.1 Baseline and Design Choices

Baseline: Our general design goal is to leverage Blockchain for securing data outsourcing. A baseline approach is to replace the cloud host by Blockchain. In the baseline, the Blockchain stores the entire dataset and directly interacts with the trusted clients of data producers and consumers through three smart

[1] In this work, the words of "compaction" and "merge" are interchangeably used.

contracts. On the write path, a "writer" contract accepts the data-write requests from the producers (encoded in the form of transactions) and sends them to the Blockchain. On the read path, a "reader" contract reads the Blockchain content to find the LSM tree level that contains the result. The Blockchain runs an offline "compaction" contract that is triggered by the same conditions of original LSM stores and that merges multiple sorted runs to reorganize the layout.

Design Space: The above baseline design raises two issues as below:

First, the baseline approach uses the Blockchain as the primary data storage, which is cost inefficient. Concretely, storing a bit in Blockchain is much more expensive and costly than storing it off-chain (e.g., in the cloud). A promising solution is to partition the LSM workflow and to result in a minimal and security-essential partition in Blockchain. This way, the primary data storage which is cumbersome is mapped off-chain to the cloud host.

Second, the baseline approach enforces a strong consistency semantic over the Blockchain which is weakly consistent; this mismatch across layers may present issues and incur unnecessary cost. More specifically, the current system of Blockchain promises only eventual consistency (or timed consistency [30]) in the sense that it allows an arbitrary delay between the transaction-submission time and the final settlement time (i.e., when the transaction is confirmed in the blockchain). The eventual consistency limits the use of Blockchain for real-time data serving and renders the baseline approach that aggressively checks the Blockchain digests to be ineffective.

3.2 Blockchain-Based TPAD Protocol

TPAD Overview: Our proposed TPAD protocol addresses the partitioning problem of an LSM tree for the minimal involvement with the Blockchain. The TPAD design separates the "data plane" (the primary data storage) and the "control plane" (e.g., digest management), and maps the former to the off-chain cloud and only loads the latter in Blockchain. Recall that an LSM tree supports three major operations (i.e., data write, read and compaction). For online data reads/writes, TPAD places only in Blockchain/smart-contract the access of the digests, while leaving data access and proof construction off-chain. To address the consistency limits, TPAD embeds the weak-consistency semantics in the application layer; for instance, it does not access the Blockchain if the results are too recent to be reflected in the Blockchain. The data-intensive computation of compaction is however materialized inside the Blockchain, which simulates a multi-client verifiable computation protocol [18]. This subsection presents the details of the TPAD protocol.

Recall that our overall system includes data producers, the cloud, the blockchain, and data consumers. The data producers generate data records and upload them to the third-party cloud-blockchain platform. Data consumers query the cloud by data keys to retrieve relevant records. For the ease of presentation, we use a concrete setting w.l.o.g. that involves two data producers, say Alice (A) and Bob (B), and one data consumer, say Charlie (C).

Initially, each data producer has a pubic-private key pair and uses the public key as her pseudonymous identity. In other words, the system is open membership that anyone can join, which is consistent with the design of open Blockchain. We assume the identities of data producer and Blockchain are established in a trusted manner, which in practice could be enforced by external mechanisms for user authentication and attestation. Conceptually, there are two virtual chains registered in the Blockchain to materialize the two states of an LSM tree, that is, the WAL and digests of data levels. These two virtual chains can be materialized in the same physical Blockchain.

On the write path, Alice, the producer, generates a record (R_A) and submits it to the Blockchain through the logger contract that logs the record as a transaction in the WAL Blockchain. The logger contract is called asynchronously in that it returns immediately and does not wait for the final inclusion of the transaction in WAL Blockchain. Simultaneously, Alice also sends the record to the untrusted cloud, which stores it in Level 0 of its local LSM system. Bob sends another record R_B to the Blockchain and cloud, which is processed in a similar fashion. The logger contract is responsible for serializing multiple records received and sending transactions in order. The total order between R_A and R_B is not resolved until the transactions are finally settled in the WAL Blockchain, which could occur as late as up to 40 min (e.g., in BitCoin) after the submission time. We maintain the consistency semantics that there is no time ordering among records in Level 0 on the cloud. Upon flush, it only flushes the records whose transactions are fully settled in the Blockchain.

On the read path, Charlie submits a query to the cloud, which returns the result as well as query proof. In addition, Charlie obtains the relevant digests from the Blockchain. Specifically, the proof consists of the Merkle authentication paths of all relevant levels, that is, the level that has the answer (i.e., membership level) and all the levels (i.e., non-membership levels) that do not have the answer but are more recent than the membership level. The digests, namely Merkle root hashes, are obtained from Digest Blockchain. As aforementioned, the system does not provide membership authentication for data in Level 0.

On the compaction path, TPAD supports two relevant contracts for data flush and merge. For the flush, the flush contract is triggered every time there is a new block found in the WAL Blockchain. It flushes all the files/records at level i to a single sorted file at Level $i + 1$. Inside the flush, the contract sorts the records at level i (which are originally organized in the time order), builds a digest of the sorted run, and sends it to the Digest Blockchain. At the same time, the off-chain cloud runs the flush computation that builds the sorted run locally.

For the merge, a separate contract merges multiple sorted runs into one run and places it at a certain level of the LSM tree. When a compaction contract runs, it validates all the input runs fed from the cloud using the digests stored in the Blockchain. It then performs the merge computation, builds a Merkle root hash on the merged run, and sends a transaction encoding the hash to update the Digest Blockchain. In the last step, the Digest Blockchain stores the digests

of different LSM levels and the contract replaces the digests by those of the merged run. At the same time, the off-chain cloud runs the merge computation that builds the sorted run locally.

The two compaction contracts update the Blockchain state and have a companion computation going on the off-chain side. Given the delay to finally settle a transaction, we defer the time the updated state in Blockchain becomes available. For instance, even though the merge contract finishes the execution and sends the transaction, the off-chain data store will wait until the transaction is settled to activate the use of merged runs. The above two compaction contracts involve data-intensive computation and are executed at off-line hours. The specific triggering conditions are described next.

The algorithms in TPAD are illustrated in Listing 2.

Compaction-Triggering Policies: In TPAD, the policy that determines when and how to run a compaction is executed by the cloud host. As aforementioned, the off-chain cloud can opt for the sized LSM tree policy where the number of files per level is fixed and an overflowing file triggers the execution of flush operation. The off-chain cloud can also take the leveled LSM tree policy where application-specific condition triggers the execution of merge operations. In practice, the sized policy lends itself to serving time-series workloads where newer data does not replace older data.

In our implementation, an LSM level in the Smart Contract program is represented by an array in memory. The output is the digest of merged data which is stored persistently on Blockchain. Note that we do not store or send the merge data in Smart Contract to save the Gas cost.

Security Analysis. We consider a data-freshness attack where an adversary, e.g., the untrusted host, presents a valid but stale key-value pair as the result. That is, given a query $\mathsf{Get}(k, ts_q)$, it returns $\langle k', v', ts' \rangle$ that belongs to the data store, while there exists another more fresh key-value record $\langle k, v, ts \rangle$ such that $ts' < ts < ts_q$.

The LPAD scheme can authenticate the following two properties that establish the data freshness: (1) Result membership: Given a result record from a specific level (called result level), the LPAD scheme can prove the membership of the record in the level using the corresponding Merkle tree. That is, given query result $\langle k', v, ts \rangle := \mathsf{Get}(k, ts_q)$, LPAD can authenticate the membership of $\langle k', v, ts \rangle$ in the level it resides in (using the per-level Merkle tree) and hence the membership in the data store. (2) Non-membership of any fresher result. That is, the LPAD scheme can prove the non-membership of any record of the same queried key in levels fresher than the result level. Note that for a given key, levels are ordered by time.

In a query-completeness attack, valid result records are deliberately omitted. The completeness security is similarly provided by the LPAD scheme with the freshness security: In LPAD, the result completeness (i.e., no valid result is missed) in each query level can be deduced from that the leaf nodes in each per-level Merkle tree is sorted by data keys.

In a forking attack, different views are presented to different querying clients (presenting "X" to Alice and "Y" to Bob). The forking-attack security (or fork consistency) can be guaranteed by LPAD by that the Blockchain can provide a single source of truth for the dataset state, and any violation (by forking) can be detected by checking the result against the Blockchain state.

4 Implementation on Ethereum

We have implemented the TPAD protocol over the Ethereum Blockchain which keeps two states: WAL and digests. The other players in the protocol, including the data producers, consumers, and the cloud, are implemented in JavaScript.

A data producer writing a record to the cloud triggers the execution of logger contract on Ethereum that computes the hash digest and sends a transaction wrapping the digest.

A data consumer submits a query by key to the cloud which returns the answer and proof. The data consumer inquires about the digests stored in the Blockchain by triggering the execution of a reader contract on Blockchain. The answer proof consists of authentication paths of Merkle trees from the cloud and is used to compare against the digests for answer verification. Note that we implement the reading of digests in a smart contract for the ease of engineering.

A compaction operation is implemented on both the cloud and Blockchain. Consider the compaction of two files (or sorted runs). First, the compaction smart-contract on the Blockchain takes as input the data stored in JSON on the cloud side and the digest hashes stored in the Blockchain. As mentioned, the compaction code validates the inputs based on the digests, conducts the merge computation by heap sort, computes the new digest of the merged run, and sends the transaction encoding the digest to the Blockchain. Second, the JavaScript program on the cloud side also runs the merge computation locally on the input files. It then replaces the input files in the local JSON store by the merged file. We choose this implementation (merge computation done on both cloud and smart contract), because the JavaScript runs the smart contract asynchronously (i.e., the call returns in JavaScript without waiting for the smart contract finishes the execution) and it saves bandwidth.

A compaction operation is implemented as a distributed process running on the both sides of cloud and Blockchain. When the cloud (or a cloud administrator) decides to merge the LSM storage, it first uploads the data to be merged to the Blockchain using a batch of transactions. Then, the cloud starts to run a local merge operation. Concurrently, the transactions sent by the cloud triggers the execution of a smart-contract that does the merge computation on the Blockchain based on the data sent earlier. The cloud and Blockchain is synchronized when the merge computations on both sides end. Concretely, the cloud, once it finishes the local merge computation, will wait until being notified by the completion event of the remote merge on the Blockchain. On implementation, the cloud merge program is written in Javascript and the synchronization is realized using Promise [9], which is a multithreading support in Javascript. After the synchronization, the cloud proceed to replace the data by the merged data.

On the blockchain, the verifiable-merge smart contract is implemented as below: The compaction code validates the input data based on the digest on Blockchain, carries out the merge computation based on heap sort, computes the new digest of the merged run, and persists it into the Blockchain by sending a transaction.

The logger contract is triggered when a data producer uploads a record and its digest. The flush contract is triggered by a block in the Blockchain is found. The compaction contract is triggered by LSM compaction policies elaborated in the next section.

Implementation Notes: The current version of Solidity (i.e., 0.4.17) does not support multi-dimensional nested array in a public function. We have to implement the array of digests as a one-dimensional array and interpret it as a two-dimensional array (by levels and files) manually in the program. To collect the Gas consumption in a view function (i.e., the function that does not change state), we call `estimateGas()` function. In our implementation, the JavaScript code runs smart contract functions through JSON ABI files generated by the truffle compiler [11]. The state overwrites in Ethereum/Solidity program has to be explicit and is realized by delete and "push" operations (Fig. 2).

```
1  TPADContract{
2    uint[] WAL;
3    unit[] digests;
4    flush(){
5      while(block_found()!=true);
6        list l0=get_6th_block();
7      validate(l0);
8      ll0=sort(l0);
9      digests.send_tx(digest(ll0));
10   }
11   compact(list l1, list l2){
12     while(l1,l2=compact_policy());
13     validate(l1,l2);
14     l12=merge(l1,l2);
15     digests.send_tx(digest(l12));
16   }
17 }
18 class Client {
19   write(record r){
20     cloud.write(r);
21     WAL.send_tx(r);
22   }
23   read(key k){
24     result a, proof p=cloud.read(r);
25     d=digests.read_tx(a);
26     if(verify(a,p,d)) return a;
27   }
28 }
```

Fig. 2. Implementing TPAD

5 Evaluation

This section presents the evaluation of TPAD. The goal is to understand the cost saving of TPAD comparing alternative designs including on-chain storage (Sect. 5.1) and other data structures (Sect. 5.2). We first present our evaluation platform.

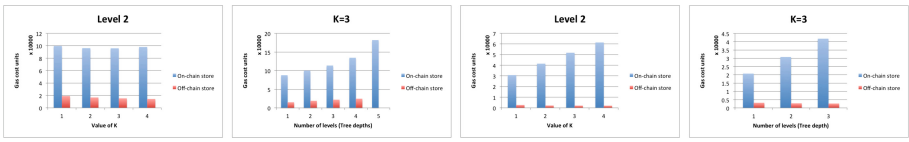
Setup: Our smart-contracts written in Solidity are compiled in the Truffle programming suit. They run on a personal Blockchain network set up by Ganache [6]. This local Blockchain network is sufficient for our evaluation purpose which only evaluates the cost consumption. For comparison, we implement the baseline approach of storing data in Blockchain. Here, the blockchain keeps a state of the LSM tree stored in a multi-dimensional storage array. In the implementation, no in-memory index is maintained and finding a record in a file is materialized by binary search. We also implement the other two baselines, namely append-only log and update-in-place structures. For the latter, we implement a binary-search tree and build a Merkle tree based on it with the root node stored in Blockchain.

5.1 Cost Saving of Off-Chain Storage

The TPAD is firstly a Blockchain logging scheme with the data stored off-chain. A relevant baseline is to treat the Blockchain as the primary storage, namely on-chain store. We implement the baseline by placing an entire LSM tree, including leaf-level data nodes, inside the Blockchain.

On our platform, we conduct experiments by driving 20,000 records into the data store. We varied the "shape" of the LSM tree in terms of the size of a level (number of files allowed in a level, K) and the number of levels. We measure the cost in terms of Gas consumption of the two approaches respectively with on-chain and off-chain storage.

The results are presented in Fig. 3. Figure 3a is the write cost when the LSM tree has two levels. With different values of K (recall K is the number of files in a level), the cost is relatively stable. Comparing the on-chain storage, the off-chain storage saves a significant amount of cost, which is about $5X$ saving. When fixing K at 3, varying the number of levels from 1 to 5, the cost of on-chain store increase which is consistent with the fact that write amplification increases



(a) Write cost at level 2 (b) Write cost with $K = 3$ (K is the number of files per level) (c) Read cost at level 2 (d) Read cost with $K = 3$

Fig. 3. On-chain storage cost versus off-chain cost

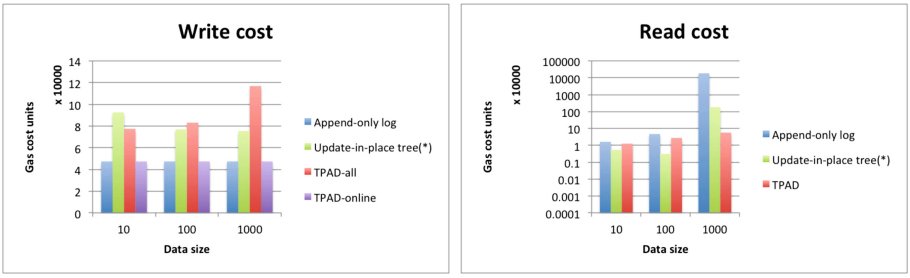
along with the number of compaction jobs. Comparing on-chain and off-chain storage, the cost saving also increases along with the number of levels. In both Figs. 3c and d, the read cost increases along with the value of K. The off-chain storage saves the Gas cost up to $60X$ and $20X$ respectively for the settings of two levels and K equal to 3.

5.2 Efficiency of LSM-Based Storage on Blockchain

The TPAD is an authenticated key-value store that supports random-access reads/writes. In this regard, relevant baselines that implement the key-value store abstraction include an append-only log where records are ordered by time and an update-in-place structure, namely a single Merkle tree where leaf nodes are ordered by keys. We implement the first baseline by simply sending the hash digest of every data write to the Blockchain. The second baseline is implemented by maintaining the root hash of the key-ordered Merkle tree in Blockchain and by translating every data read/write to a leaf-to-root path traversal on the Merkle tree. In more details, a data read to the cloud store would present as a proof the authentication path of the leaf node to the root hash of the Merkle tree and a data write consists of a data read followed by a local modification and a remote update to the authentication path.

We conduct small-scale experiments by loading a thousand records into the storage system; the keys and values in the records are randomly distributed. We measure the average costs of read and write. The cost consists of the Gas cost for running smart contract that retrieves the digests stored in the Blockchain and the costs of preparing and verifying query proof (e.g., the authentication paths in Merkle trees). We use a heuristic to combine the two costs by multiplying the Gas cost by 100 times before adding it and the proof-related cost. The proof-related cost is measured by the number of cycles spent locally for proof verification.

The results are presented in Fig. 4. The results show that the TPAD can result in cost efficiency on both reads and writes. Concretely, for the write results in Fig. 4a, the online part of TPAD has a similar cost with the other two baselines, as each write results in a single transaction in all three approaches. The



(a) Write cost

(b) Read cost

Fig. 4. LSM tree-based TPAD compared against other structures in cost

overall TPAD approach that includes both online and offline operations (i.e. compaction) would incur write amplification as shown in Fig. 4a. For the data read results in Fig. 4b, the cost saving of TPAD is significant, provided that the y-axis is plotted in log scale. The TPAD incurs even lower cost than update-in-place trees partly because of the locality in our query workloads where recently updated data is more likely being queried.

6 Discussion: Data Confidentiality and Key Management

Data producers concerned about data confidentiality can upload the records in an encrypted form. Specifically, a data producer sends the ciphertext of the record, instead of plaintext, to the third-party host. The decryption key is shared through an offline key-distribution channel between the data producer and the data consumers who are permissioned to access the record. Those consumers can obtain the ciphertext of the record from the host and use the key to decrypt. To enable the query over ciphertext, we consider the use of deterministic encryption which supports exact-match query in the encrypted form, that is, the consumer could submit the encrypted query key to the host who will conduct exact-match query between the query ciphertext and data ciphertext. The integration with more secure encryption primitives is complementary to this scope of this work.

The data-encryption layer is laid over the membership-/data- authentication layer of TPAD (as described above). This is similar to the classic encryption-then-authentication scheme [20]. With deterministic encryption, the merge operation of TPAD occurs in the domain of ciphertext.

7 Related Work

7.1 Blockchain Applications

A common paradigm of supporting applications over Blockchain is that the application-level workflow is partitioned and mapped to the on-/off-chain parts. Decentralizing privacy [34] supports access-control oriented data-sharing applications over Blockchain. It publishes the access control list onto the Blockchain and enforces the access control by smart contract. A similar approach is used in MedRec [13] to enforce access control for medical data sharing. MedRec runs a proprietary Blockchain network where miners are computers in an academic environment and are rewarded by an anonymized medical dataset.

Namecoin [19] and Blockstack [12] support general-purpose key-value storage in the decentralized fashion. They allow open-membership and accept any users to upload their data signed with their secret keys. They support the storage of name-value binding, with a canonical application to be DNS servers. Namecoin is a special-purpose Blockchain system and Blockstack is realized as a middleware on top of any Blockchain substrates. The VirtualChain in Blockstack supports a (single) state-machine abstraction. Re-purposing original Blockchain for storage,

its system design tackles the challenges of limited storage capacity, long write latency, and low transactional throughput.

Catena [31] is probably the closest related work to TPAD. Catena is a non-equivocation scheme over the Blockchain that repurposes its no-double-spending security for non-equivocation in logging and auditing. In essence, it aligns the application-specific log (for auditing) with the underlying linear Blockchain and reuses the non-fork property of Blockchain for the non-fork application log. Briefly, Catena's mechanism is to build a virtual chain on BitCoin blockchain by (ab)using `OP_RETURN` transaction interface. Logging sends a BitCoin transaction and auditing performs an ordered sequence of statement-verification calls in the log history. The statement verification does not scan full history but simply runs the Bitcoin-validation logic (e.g., Simplified Payment Validation), which ensures no BitCoin double-spending. Importantly, it enforces the rule that a Catena transaction spends the output of its immediate predecessor for efficient validation. The genesis transaction is served as the ground truth of validation and it assumes a broadcast channel to establish the consistent view of the Genesis transaction.

Our TPAD is different from Catena in the following senses: (1) Catena is built on Bitcoin or the first-generation blockchain, and TPAD leverages the smart-contract capabilities widely existing in the latest Blockchain systems, such as Ethereum [4]. (2) More importantly, Catena only supports auditing which is essentially sequential reads. TPAD supports verifiable random-reads. (3) While Catena claims to be low cost, the increasing rate of BitCoin ($700 per BitCoin at the time of Catena paper writing versus $17000 per BitCoin at early 2018) makes the Catena more expensive. TPAD address the cost minimization of these repurposed Blockchains.

7.2 Outsourced Storage and ADS

Outsourcing data storage to a third-party host such as public cloud is a popular application paradigm. In the presence of an untrusted host, it is important to ensure the data security, especially membership authenticity. An authenticated data structure (ADS) is a protocol that formally the security property. Depending on the operations supported (queries and updates), an ADS protocol can be constructed by different cryptographic primitives such as secure hash and Merkle trees [22], SNARK [14], bilinear pairings [32,33], etc.

8 Conclusion

This work proposes the TPAD system for securely outsourcing data storage on third-party hosts by leveraging the Blockchain. Instead of using Blockchain as a time-ordered log, TPAD lays the log-structured merge tree (LSM tree) over the Blockchain for efficient and lightweight logging. Realizing the design, a small

state is persisted in the Blockchain and computation-oriented compaction operations are carried out by smart contracts. With the implementation in Ethereum/-Solidity, we evaluate the proposed logging scheme and demonstrate its performance efficiency and effectiveness in cost saving.

References

1. Apache Cassandra. http://cassandra.apache.org/
2. Apache HBase. http://hbase.apache.org/
3. Bitcoin. https://bitcoin.org/en/
4. Ethereum project. https://www.ethereum.org/
5. Facebook RocksDB. http://rocksdb.org/
6. Ganache. http://trufflframework.com/ganache/
7. Google LevelDB. http://code.google.com/p/leveldb/
8. Litecoin. https://litecoin.org/
9. Promise. https://developer.mozilla.org/en-us/docs/web/javascript/reference/global_objects/promise
10. Solidity. https://solidity.readthedocs.io/en/develop/
11. Truffle. http://trufflframework.com/
12. Ali, M., Nelson, J.C., Shea, R., Freedman, M.J.: Blockstack: a global naming and storage system secured by blockchains. In: Gulati, A., Weatherspoon, H. (eds.) 2016 USENIX Annual Technical Conference, USENIX ATC 2016, Denver, CO, USA, 22–24 June 2016, pp. 181–194. USENIX Association (2016)
13. Azaria, A., Ekblaw, A., Vieira, T., Lippman, A.: MedRec: using blockchain for medical data access and permission management. In: Awan, I., Younas, M. (eds.) 2nd International Conference on Open and Big Data, OBD 2016, Vienna, Austria, 22–24 August 2016, pp. 25–30. IEEE Computer Society (2016)
14. Ben-Sasson, E., Chiesa, A., Genkin, D., Tromer, E., Virza, M.: SNARKs for C: verifying program executions succinctly and in zero knowledge. In: Canetti, R., Garay, J.A. (eds.) CRYPTO 2013. LNCS, vol. 8043, pp. 90–108. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-40084-1_6
15. Chang, F., et al.: Bigtable: a distributed storage system for structured data (awarded best paper!). In: OSDI, pp. 205–218 (2006)
16. Chung, H., Iorga, M., Voas, J.M., Lee, S.: Alexa, can I trust you? IEEE Comput. **50**(9), 100–104 (2017)
17. Elmasri, R., Navathe, S.B.: Fundamentals of Database Systems, 2nd edn. Benjamin/Cummings, Redwood City (1994)
18. Gordon, S.D., Katz, J., Liu, F.-H., Shi, E., Zhou, H.-S.: Multi-client verifiable computation with stronger security guarantees. In: Dodis, Y., Nielsen, J.B. (eds.) TCC 2015. LNCS, vol. 9015, pp. 144–168. Springer, Heidelberg (2015). https://doi.org/10.1007/978-3-662-46497-7_6
19. Kalodner, H.A., Carlsten, M., Ellenbogen, P., Bonneau, J., Narayanan, A.: An empirical study of namecoin and lessons for decentralized namespace design. In: 14th Annual Workshop on the Economics of Information Security, WEIS 2015, Delft, The Netherlands, 22–23 June 2015 (2015)
20. Katz, J., Lindell, Y.: Introduction to Modern Cryptography. Chapman and Hall/CRC Press (2007)
21. Li, F., Hadjieleftheriou, M., Kollios, G., Reyzin, L.: Dynamic authenticated index structures for outsourced databases. In: SIGMOD Conference, pp. 121–132 (2006)

22. Merkle, R.C.: Protocols for public key cryptosystems. In: IEEE Symposium on Security and Privacy, pp. 122–134 (1980)
23. Narayanan, A., Bonneau, J., Felten, E.W., Miller, A., Goldfeder, S.: Bitcoin and Cryptocurrency Technologies - A Comprehensive Introduction. Princeton University Press, Princeton (2016)
24. O'Neil, P.E., Cheng, E., Gawlick, D., O'Neil, E.J.: The log-structured merge-tree (LSM-tree). Acta Inf. **33**(4), 351–385 (1996)
25. Papamanthou, C., Shi, E., Tamassia, R., Yi, K.: Streaming authenticated data structures. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 353–370. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-38348-9_22
26. Papamanthou, C., Tamassia, R., Triandopoulos, N.: Authenticated hash tables based on cryptographic accumulators. Algorithmica **74**(2), 664–712 (2016)
27. Rosenblum, M.: The Design and Implementation of a Log-Structured File-System. Kluwer, Norwell (1995)
28. Silberschatz, A., Korth, H.F., Sudarshan, S.: Database System Concepts, 5th edn. McGraw-Hill Book Company, Boston (2005)
29. Tamassia, R.: Authenticated data structures. In: Di Battista, G., Zwick, U. (eds.) ESA 2003. LNCS, vol. 2832, pp. 2–5. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-39658-1_2
30. Terry, D.: Replicated data consistency explained through baseball. Commun. ACM **56**(12), 82–89 (2013)
31. Tomescu, A., Devadas, S.: Catena: efficient non-equivocation via bitcoin. In: 2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, 22–26 May 2017, pp. 393–409. IEEE Computer Society (2017)
32. Zhang, Y., Katz, J., Papamanthou, C.: IntegriDB: verifiable SQL for outsourced databases. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, 12–16 October 2015, pp. 1480–1491 (2015)
33. Zhang, Y., Katz, J., Papamanthou, C.: An expressive (zero-knowledge) set accumulator. In: 2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, 26–28 April 2017, pp. 158–173. IEEE (2017)
34. Zyskind, G., Nathan, O., Pentland, A.: Decentralizing privacy: using blockchain to protect personal data. In: 2015 IEEE Symposium on Security and Privacy Workshops, SPW 2015, San Jose, CA, USA, 21–22 May 2015, pp. 180–184. IEEE Computer Society (2015)